Towards A High-Level Implementation of Flexible Parallelism Primitives for Symbolic Languages

Amadeo Casas[1] Manuel Carro[2] Manuel Hermenegildo[1,2]
[1]University of New Mexico
Albuquerque, NM (USA)
{amadeo,herme}@{cs,ece}.unm.edu
[2]Universidad Politecnica de Madrid
Madrid (Spain)
{mcarro,herme}@fi.upm.es

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and Logic Languages; Concurrent, Distributed, and Parallel Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*

General Terms

Performance, Languages

Keywords

Parallelism, logic programming, symbolic computation

1. INTRODUCTION

The advent of multicore processors is bringing renewed interest in parallelism and, accordingly, in the development of languages and tools to simplify the task of writing parallel programs. This is especially important for the complex, non-regular algorithms often found in software which performs non-trivial symbolic tasks. Such software can benefit from being written in a high-level language whose nature is symbolic as well, since this narrows the gap between the conceptual definition of the task to be performed and the code which executes it. In our case we will use for concreteness a logic-based multiparadigm language, Ciao [1], which is based on a logic-programming kernel and a flexible mechanism whereby multiple extensions are built supporting Prolog, functional programming, constraint programming, and other system- and user-level languages. The base language and system features dynamic typing, higher-order capabilities, polymorphism, and static type inference and checking (also of non-trivial properties, such as computational complexity). Such language capabilities are largely orthogonal to parallelism; however, the way parallelism is expressed combines seamlessly with the the rest of the language.

An advantage of logic-based languages (and, in general, of declarative languages) is that their clean semantics and high-level nature makes it possible to perform automatic parallelization more easily [4, 2]. At the same time, the runtime system often has more degrees of freedom to decide how parallel tasks are to be scheduled. A wealth of research on parallel execution of logic programs has been reported so far [4]. Two main forms of parallelism have been identified and exploited: *or-parallelism* and *and-parallelism*. The former tries to explore in parallel branches of any search performed by the program, while the latter (on which we will focus) is aimed at executing parts of general computations (*conjunctions* of goals) in parallel. While or-parallelism is only useful when there is search involved, and-parallelism arises additionally in many classes of applications, with *divide-and-conquer* and *map-style* algorithms being classical examples.

Most implementations of and-parallelism rely on complex low-level machinery [4]. The alternative approach we propose (in fact an evolution of [5]) is based on raising the implementation of certain components to the level of the source language while keeping only some selected operations (related to thread handling, locking, etc.) at a lower level. This approach does not eliminate altogether modifications to the abstract machine, but it greatly simplifies them. We expect this separation of concerns to make it possible to easily explore variations on execution schemes, such as goal scheduling, supporting sophisticated goal dependencies, etc., to better suit the application at hand. Also, it allows the implementation language to be used as a basis for designing and compiling domain-specific user languages (such as, e.g., describing strategies for theorem provers) with parallelism in mind.

2. FLEXIBLE PARALLELISM PRIMITIVES

Some well-known and successful and-parallel systems [5] use the parallel conjunction operator `&/2`, instead of the sequential *comma* "," to express fork-join (nested) parallelism:

EXAMPLE 1. *The code below is a parallel program which symbolically derives arithmetic expressions:*

```
deriv(U+V,X,DU+DV):-            !, deriv(U,X,DU) & deriv(V,X,DV).
deriv(U-V,X,DU-DV):-            !, deriv(U,X,DU) & deriv(V,X,DV).
deriv(U*V,X,DU*V+U*DV):-        !, deriv(U,X,DU) & deriv(V,X,DV).
deriv(U/V,X,(DU*V-U*DV)/V^2):- !, deriv(U,X,DU) & deriv(V,X,DV).
deriv(-U,X,-DU):-               !, deriv(U,X,DU).
deriv(exp(U),X,exp(U)*DU):-     !, deriv(U,X,DU).
deriv(log(U),X,DU/U):-          !, deriv(U,X,DU).
deriv(U^N,X,DU*N*U^N1):-        !, integer(N), N1 is N-1,
                                   deriv(U,X,DU).
deriv(X,X,1):-                  !.
deriv(_C,_X,0).
```

In our approach we use however more flexible constructions to represent parallelism by using two operators, `&>/2` and `<&/1`, defined as follows [3]: `G &> H` schedules `G` for parallel execution and continues executing the code after `G &>`

PASCO'07, July 27–28, 2007, London, Ontario, Canada.

Bench.	1	2	3	4	5	6	7	8
AI-AKL	0.91	1.73	1.67	1.64	1.65	1.63	1.63	1.61
Ann	0.96	1.83	2.67	3.45	4.19	4.91	5.61	6.28
Boyer	0.15	0.29	0.43	0.55	0.66	0.73	0.78	0.82
BoyerGC	0.82	1.65	2.44	3.10	3.56	4.19	4.55	5.08
Deriv	0.14	0.27	0.40	0.52	0.64	0.75	0.85	0.90
DerivGC	0.82	1.58	2.30	2.98	3.59	4.15	4.72	5.17
Fib	0.16	0.31	0.46	0.62	0.78	0.92	1.07	1.21
FibGC	0.99	1.98	2.97	3.96	4.93	5.92	6.90	7.89
Hanoi	0.46	0.88	1.22	1.56	1.80	2.10	2.32	2.47
HanoiGC	0.85	1.61	2.13	2.65	2.97	3.35	3.62	3.81
MSort	0.59	1.11	1.29	1.75	1.94	2.19	2.36	2.54
MSortGC	0.89	1.62	1.94	2.58	2.60	2.92	2.94	3.32
MMatrix	0.81	1.61	2.37	3.11	3.70	4.58	5.27	5.77
QSort	0.58	1.13	1.57	2.02	2.36	2.70	2.84	3.04
QSortGC	0.97	1.86	2.51	2.91	3.44	3.48	3.63	3.78

Table 1: Speedups (1 to 8 processors). Sequential execution corresponds to Speedup = 1.0.

`H`. `H` is a *handler* which contains (or *points to*) the state of goal `G`. `H <&` waits for the goal associated to `H` to finish. After that point the final bindings made by `G` to its variables are available to the executing thread.

In our current implementation for shared-memory multiprocessors, each agent (processor + virtual machine) executes a sequential Prolog virtual machine (with negligible overhead imposed on the sequential parts) extended with a goal stack where pointers to the generated parallel goals are pushed. If `G` has finished, `H <&` immediately succeeds and the bindings made by `G` are available. If `G` has not been taken by any other agent, it is executed locally, and then `H <&` succeeds. If `G` has been taken but it has not finished yet, then the executing agent will repeatedly try to run some other goal available. If none is available, execution suspends until there is some goal available, or until `G` finishes. With the previous definitions, the `&/2` operator can be simply written as G_1 `&` G_2 `:-` G_2 `&> H, call(`G_1`), H <& .`

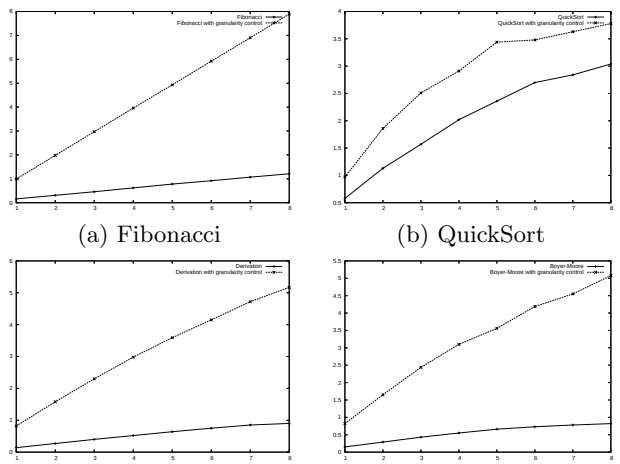
The `&>/2` and `<&/1` operators do not assume any particular architecture, and hence they can be also implemented in distributed memory machines. Specialized versions are also available, to create agents that execute goals "on demand" or to adapt to the (very common) deterministic case. Most importantly, using the `&>/2` and `<&/1` primitives, dependency graphs other than fork-join can be expressed, and more parallelism may be exploited.

3. EXPERIMENTAL RESULTS

We have performed a preliminary evaluation of the system performance using a series of classic benchmarks for (independent) and-parallelism. All the results were obtained by averaging ten runs on a Sun Fire T2000 with 8 cores, with 4 threads each, 8 Gb of memory, and running Solaris. Speedups with respect to the sequential execution (using from 1 to 8 processors)[1] are presented in Table 1 and (for some selected benchmarks) in Figure 1.

The programs used for benchmarking perform mostly symbolic computations. Two of them (**Deriv** and **Boyer**) can be considered examples of symbolic mathematics. Summarizing, **AI-AKL** is part of an analyzer for the AKL language; **Ann** is the parallelized version of one of the &-Prolog parallelizers; **Boyer** is a reduced version of the Boyer-Moore theorem prover; **Deriv** calculates the derivative of an expression; **Fib** is the doubly recursive Fibonacci function;

[1]Memory-intensive benchmarks obtain, in our experience, sublinear speedups if the number of agents exceeds the number of core processors, even for independent computations.



(a) Fibonacci (b) QuickSort

(c) Derivation (d) Boyer-Moore

Figure 1: Speedups with and without gran. control.

Hanoi computes a solution to the Towers of Hanoi; **MSort** and **QSort** sort a list using the mergesort and quicksort algorithms; and **MMatrix** multiplies two matrices. The **GC** versions perform granularity control.

We observe that reasonable speedups are achievable, but the additional overhead in the current prototype implementation (mainly due to the lifting of parallelism-related primitives to the source language level) makes it advisable to use granularity control (Figure 1). While this complicates the code, automatic compile-time granularity control [6] can be applied to alleviate the burden of adding such control by hand. This, together with automatic compile-time parallelization [2], often makes it possible to write (sequential) code which matches closely the high-level algorithm and to obtain speedups automatically. We conclude that the results are quite reasonable given the simplicity of our implementation approach and encourage us to work further on the optimization of our high-level implementation.

4. REFERENCES

[1] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). `http://www.ciaohome.org`.

[2] F. Bueno, M. G. de la Banda, M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization. *ACM TOPLAS*, 21(2):189–238, 1999.

[3] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *AGP'96*, pages 67–78, 1996.

[4] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, July 2001.

[5] M. Hermenegildo, K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[6] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.